THE UNIVERSITY OF NEW MEXICO

ECE 538
Advanced Computer Architecture

Instructor: Lei Yang

Department of Electrical and Computer Engineering

September 29, 2021

Instruction Set Architecture（ISA）Design

Referred to Appendix K in textbook

“Computer Architecture: A Quantitative Approach”

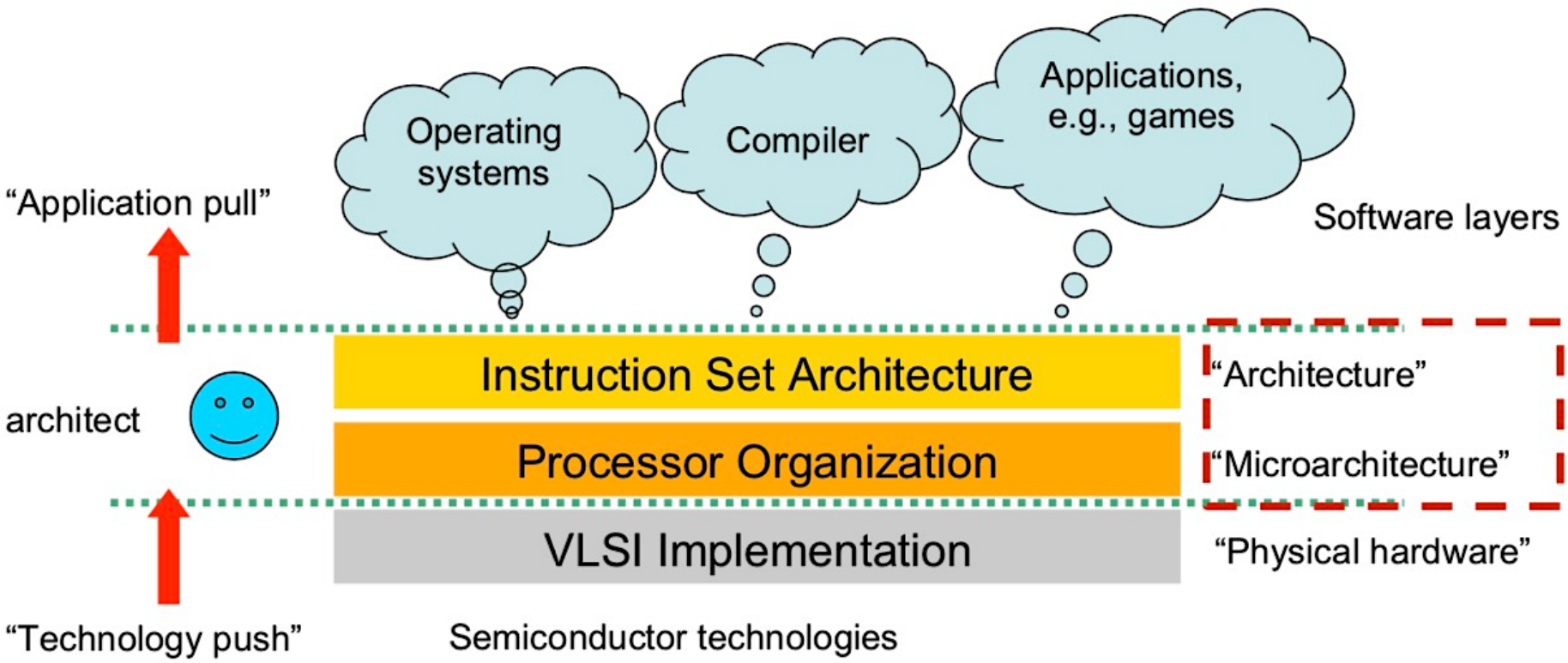
What is Computer Architecture?

- Computer architecture deals with design and implementation of computer hardware. There are three main subcategories:

  - *Instruction Set Architecture, or ISA*:

    ISA defines the machine code that a processor reads and acts upon as well as the word size, memory address modes, processor registers, and data type.

What is Computer Architecture?

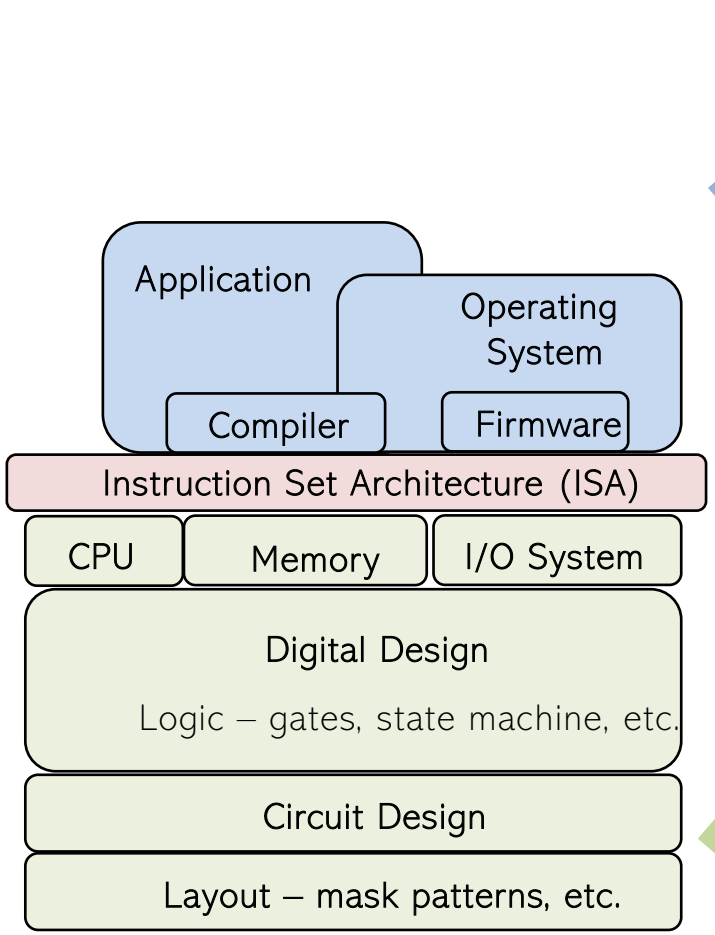


(Click to watch the video)

Instruction Set Architecture (ISA)

- Instruction Set Architecture (ISA): A set of following specifications which a programmer must know to write a correct/efficient program for specific machine.
  - Instruction format
  - Length of instruction and size of field
  - Word size :8-bit, 16-bit or 32-bit etc.
  - Set of all operations: opcodes/ machine language
  - Register file specification: size, width, and its usage of registers in CPU
  - Memory address space and addressability: no. of addressable locations & bits per location
  - Addressing modes: ways of specifying and accessing operand(s): indicates how an address (memory or register) is determined
  - Operand locations: all in registers, register and memory or all in memory

Instruction Set Architecture (ISA)



Software

High-level Language Program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

Compiler

Interface Between HW/SW

Assembly Language Program (for MIPS)

```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```

Assembler

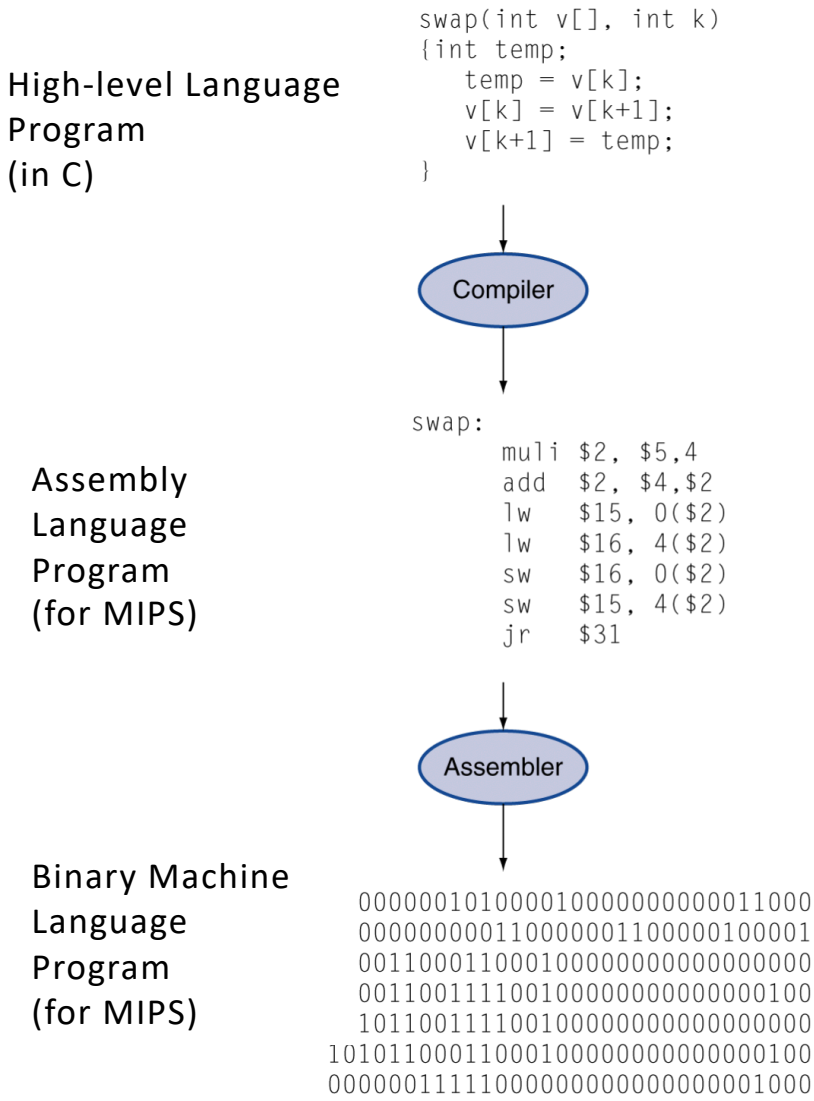
Hardware

Binary Machine Language Program (for MIPS)

```
00000000101000010000000000011000
00000000000011000000110000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

ISA – Hardware/Software Interface

- High-level language
  - – Level of abstraction closer to problem domain
  - – Provides for productivity and portability
- Assembly language
  - – Textual representation of instructions
- Hardware representation
  - – Binary digits (bits)
  - – Encoded instructions and data



Instruction Set Architecture (ISA)

- Instruction Set
  - The repertoire of instructions of a computer
  - Different computers shave different instruction sets
    - But with many aspects in common
  - Early computers had very simple instruction sets
    - Simplified implementation
  - Many modern computers also have simple instruction sets
- Necessary Operations
  - Detailed design of the hardware components is driven by choice of operations in instruction set.

Instruction Set Architecture (ISA)

- Types of Instructions
  - Data transfer instructions
    - Data transfer and storage
  - Data manipulation instructions
    - Capabilities of the processor
  - Program sequencing and control instructions
    - Sequential flow of instructions
  - Input and output instructions
    - Perform I/O transfers

Instruction Set Architecture (ISA)

- *Why More Instructions?*
  1. Programmer convenience
     (Reduce program development time)
  1. Performance improvement
     (Reduce program execution time)

Note:
modern optimizing compilers remove reason #1 because general users' program only in HLL. Only the compiler writer needs to use the instruction set directly! (Even operating systems are written in HLL, e.g. C. was developed to write UNIX)

- What else Instructions are needed?
  - Instructions are chosen for ability to improve program execution performance.
  - Performance depends most on instructions most frequently used.
  - Moral: only a small number of instruction types are executed most of the time!

	MIPS int	MIPS FP
arithmetic and logic	43%	51%
data transfer (load/store)	37%	44%
control (branch/jump)	20%	4%

THE UNIVERSITY OF NEW MEXICO

ECE 538
Advanced Computer Architecture

Instructor: Lei Yang

Department of Electrical and Computer Engineering

September 29, 2021

2021-09-29: End Class 11 here

- ISA in computer architecture
  - ✓ Interface between HW/SW
  - ✓ Types of ISA
  - ✓ Functionalities

MIPS ISA: A Design Example

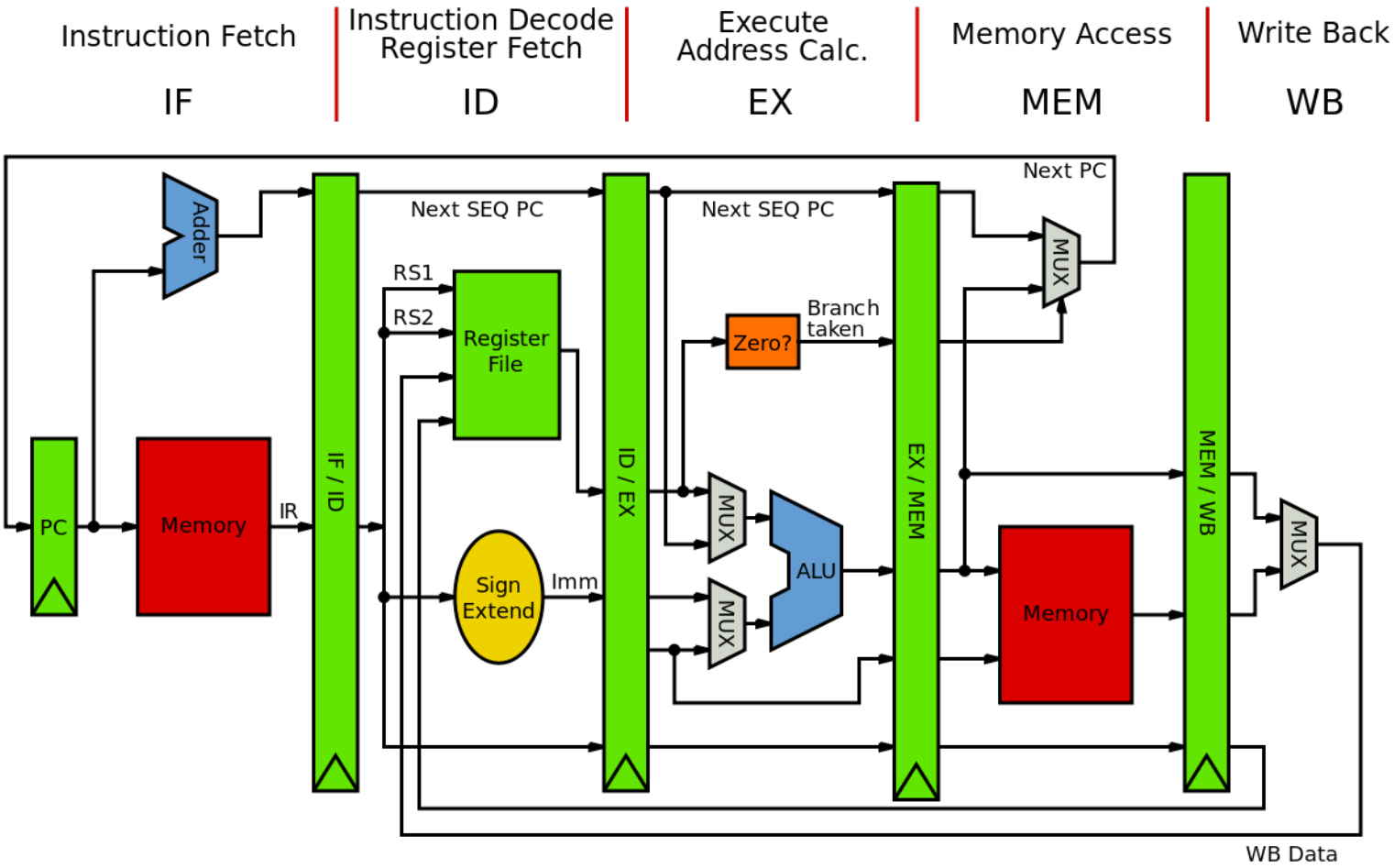
Memory and register specification, Instruction format,

addressing modes and instruction set.

- MIPS
  - Developed by John Hennessy and his colleagues at Stanford in the 1980's.
  - Used in many commercial systems: Silicon Graphics, Nintendo, Cisco
  - Simple, sensible, regular, widely used RISC architecture
  - Reduced Instruction Set Architecture (RISC) is widely used for its simpler implementation, easier pipelining, and superscalar computing
    - DEC alpha, PowerPC (Mac, IBM servers);
    - SPARC (Sun)
    - ARM processors (smartphones and embedded systems)
    - Many from Intel, AMD, and Atmel, etc.
    - Two revisions of MIPS at present: MIPS32 for 32-bit implementation and MIPS64 for 64-bit implementation.

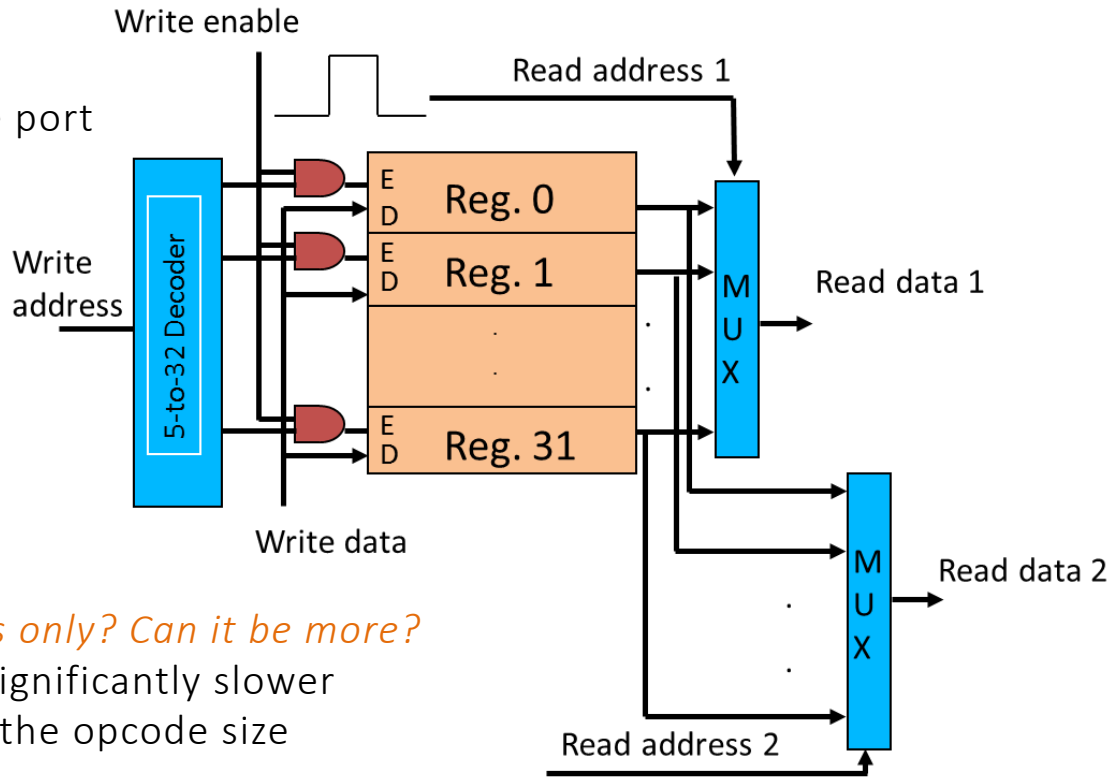
MIPS ISA: A Design Example

- MIPS (Microprocessor without Interlocked Pipeline Stages):



MIPS ISA - Register specification

- CPU Registers:
- Simple, sensible, regular, widely used RISC architecture
- General purpose register (GPR) file
  - ✓ Contains 32 32-bit registers
  - ✓ Contains 2 read ports, 1 write port



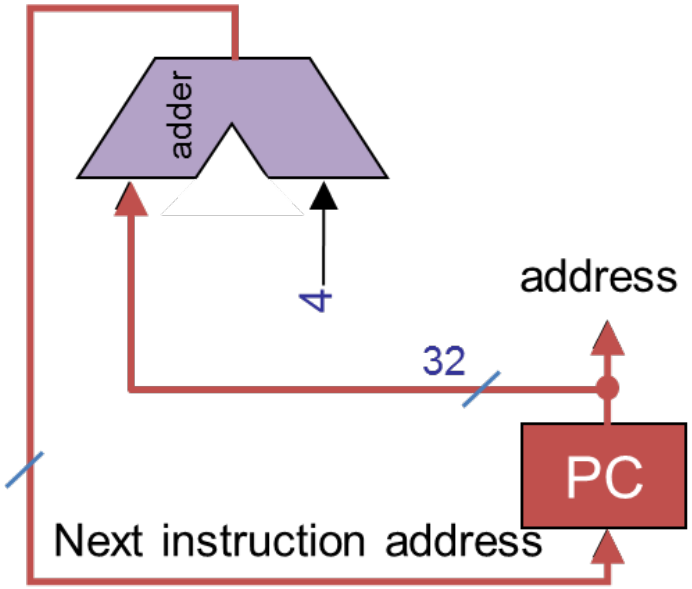
Q: Why only thirty-two registers only? Can it be more?

A1: More registers will make it significantly slower

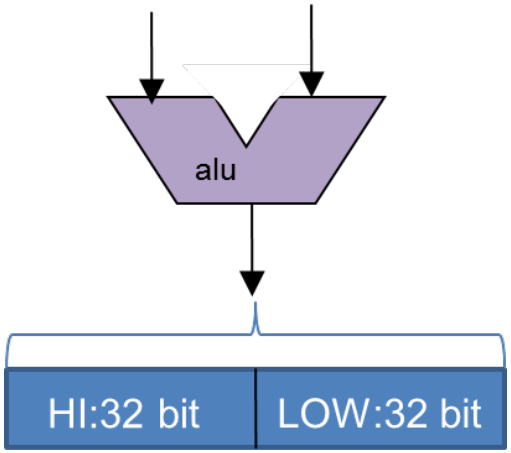
A2: More registers will increase the opcode size

MIPS ISA - Register specification

- Program Counter (PC):
  - 32-bit register that holds the address of the next instruction



- Accumulator register (Acc)
  - 64-bit register that holds the ALU output



64 bit Acc

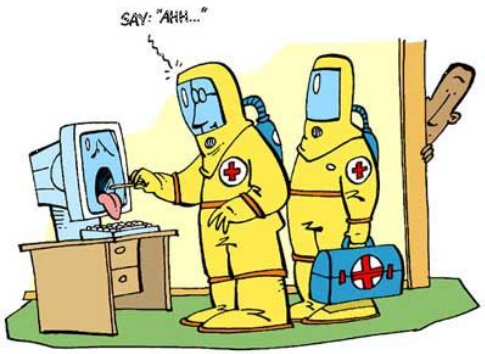
A status register

MIPS ISA - Register specification

- MIPS Register Usage

Name	Register #	Usage	Preserved on call?
$zero	0	the constant value 0	n.a.
$at	1	reserved for assembler	
$v0–$v1	2 – 3	return values of functions and expression evaluation	no
$a0–$a3	4 – 7	function arguments/parameters	yes
$t0–$t7	8 – 15	temporaries, caller saved register	no
$s0–$s7	16 – 23	saved values, caller saved registers	yes
$t8–$t9	24 – 25	more temporaries	no
$k0–$k1	26 – 27	Kernel use : reserved for OS	
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address	yes

- Each register has 32-bit (1 word)- 4 Bytes (1 byte= 8 bits)
- Temporaries are not preserved across procedure calls, while saved values are preserved across procedure calls.



MIPS ISA - Register specification

- Memory Organization
  - 32-bit addresses bus of MIPS processor
  - MIPS memory is byte-addressable: each memory address references 1 byte
  - Hence with 32-bit addresses the processor can use 2^{32} bytes = 4 GB
  - In 32-bit MIPS processor each word consists of 32 bits, i.e.

    Word length = 32 bits = 4 bytes

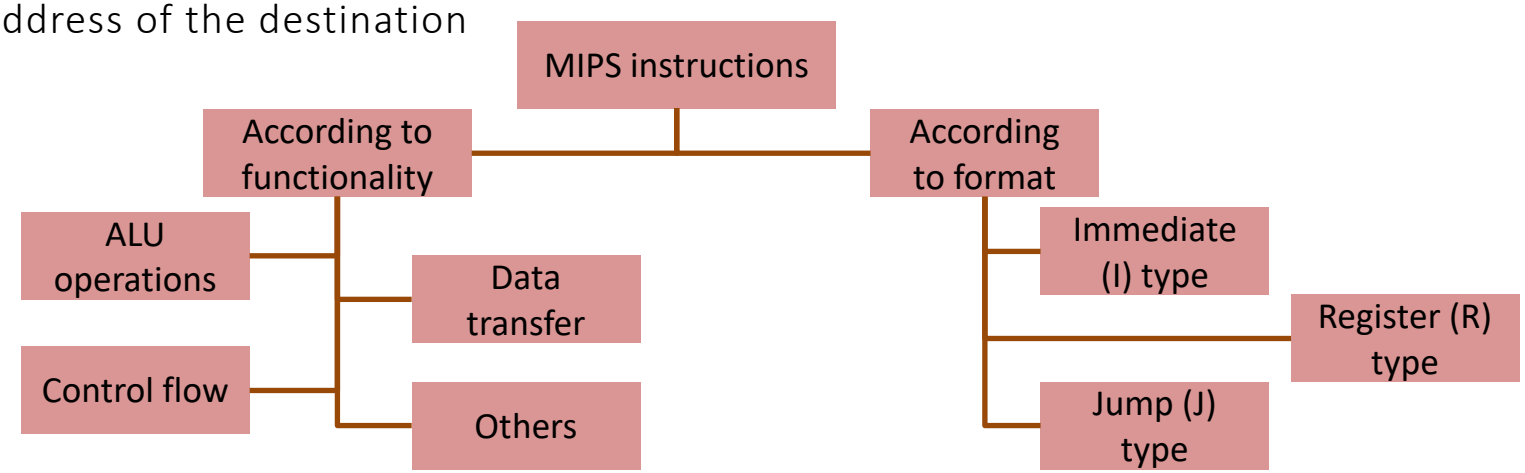
Address	Data
0x00000000	0xAC
0x00000001	0x1C
0x00000002	0xAB
0x00000003	0x5E
0x00000004	0x55
……	……
0xFFFFFFFC	0x62
0xFFFFFFFD	0x03
0xFFFFFFFE	0xD3
0xFFFFFFFF	0x5A

MIPS ISA - Register specification

- Instruction Format
  - Each MIPS instruction is 32 bits and it has 2 fields: Opcode and Operands
  - *Opcode*: Portion of a machine language instruction that specifies the operation to be performed. (add, sub, or load, and store etc.)

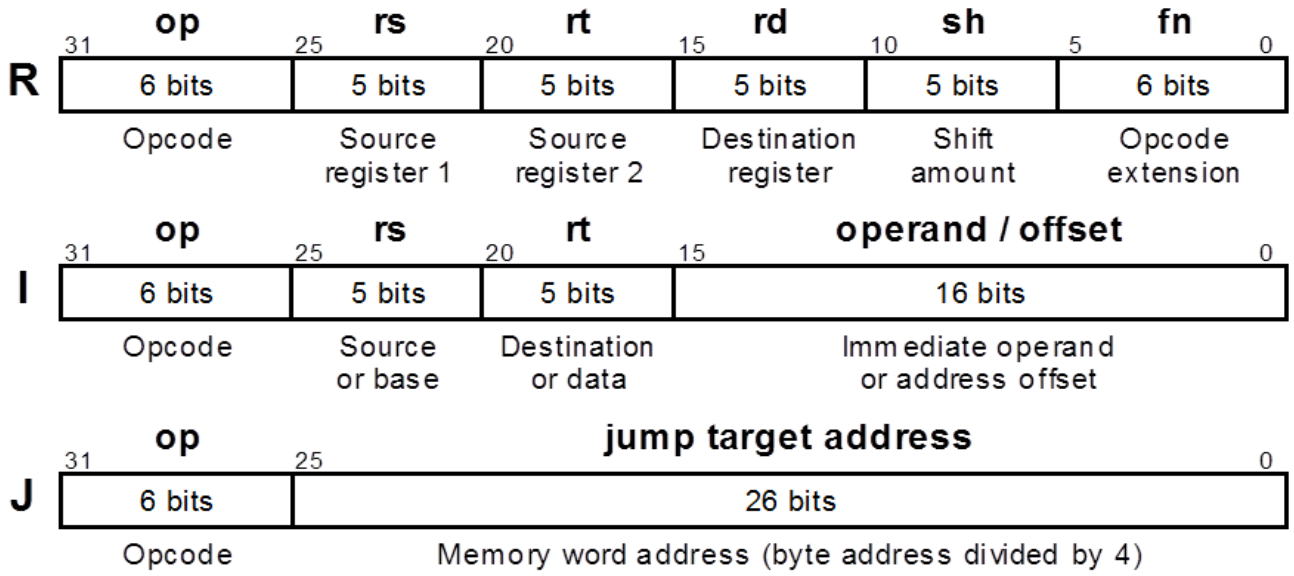
6 bits	26 bits
Opcode	Operand

  - *Operand*: Part of computer instruction that may contain
    - Input and output data (source and destination)
    - Address of the destination



MIPS ISA

- Classification of Instructions based on Instruction Format
  - R-type or (Register Format)- All data values (operands/results) are located in registers
  - I-type or (Immediate format)- One of the operand is an Immediate value and another located in register
  - J-type or (Jump format)- for jump or branch instructions



- rs represents source register
- rd represents destination register
- rt can be either source or destination depending on instruction format

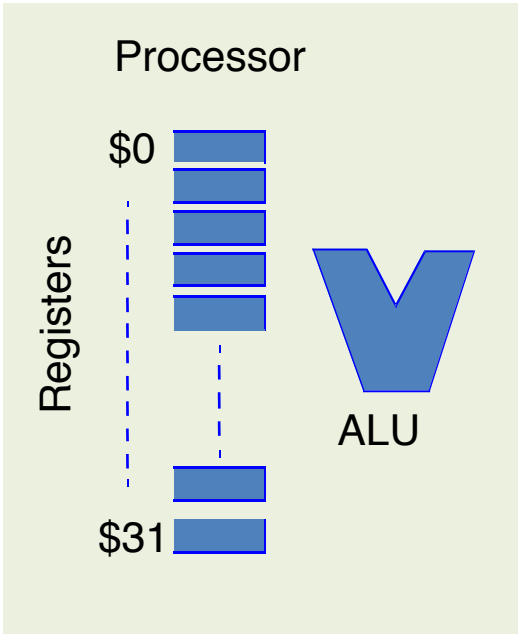
MIPS ISA

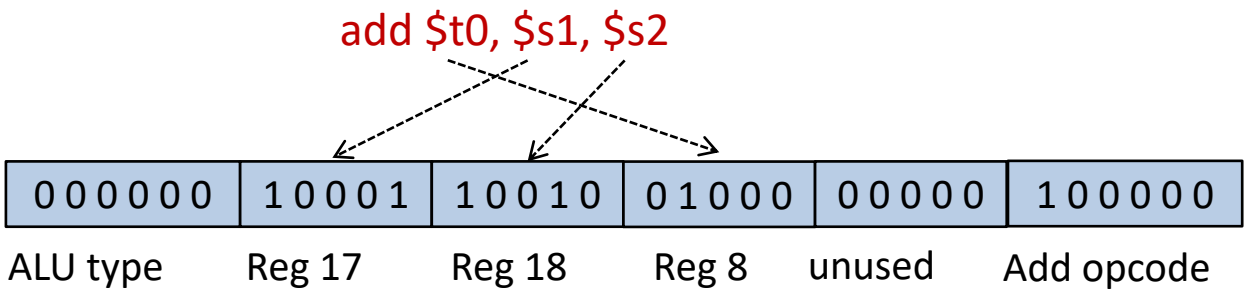
- C statement: $f = (g + h) - (i + \mathrm{j})$
- MIPS instructions

```
# add t0, g, h
# add t1, i, j
# sub f, t0, t1
```

- Opcode/mnemonic, operands, source/destination
- Operands must be registers, not variables

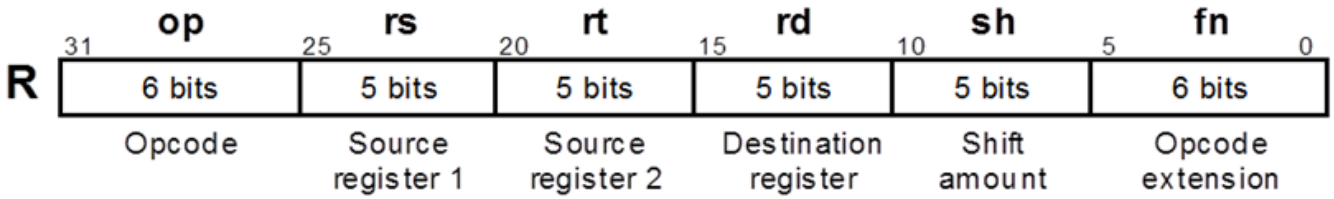
```
# add $8, $17, $18 (add $t0, $s1, $s2)
# add $9, $19, $20 (add $t1, $s3, $s4)
# sub $16, $8, $9  (sub $s0, $t0, $t1)
```





MIPS ISA – Instruction Format

- Register (R) Type

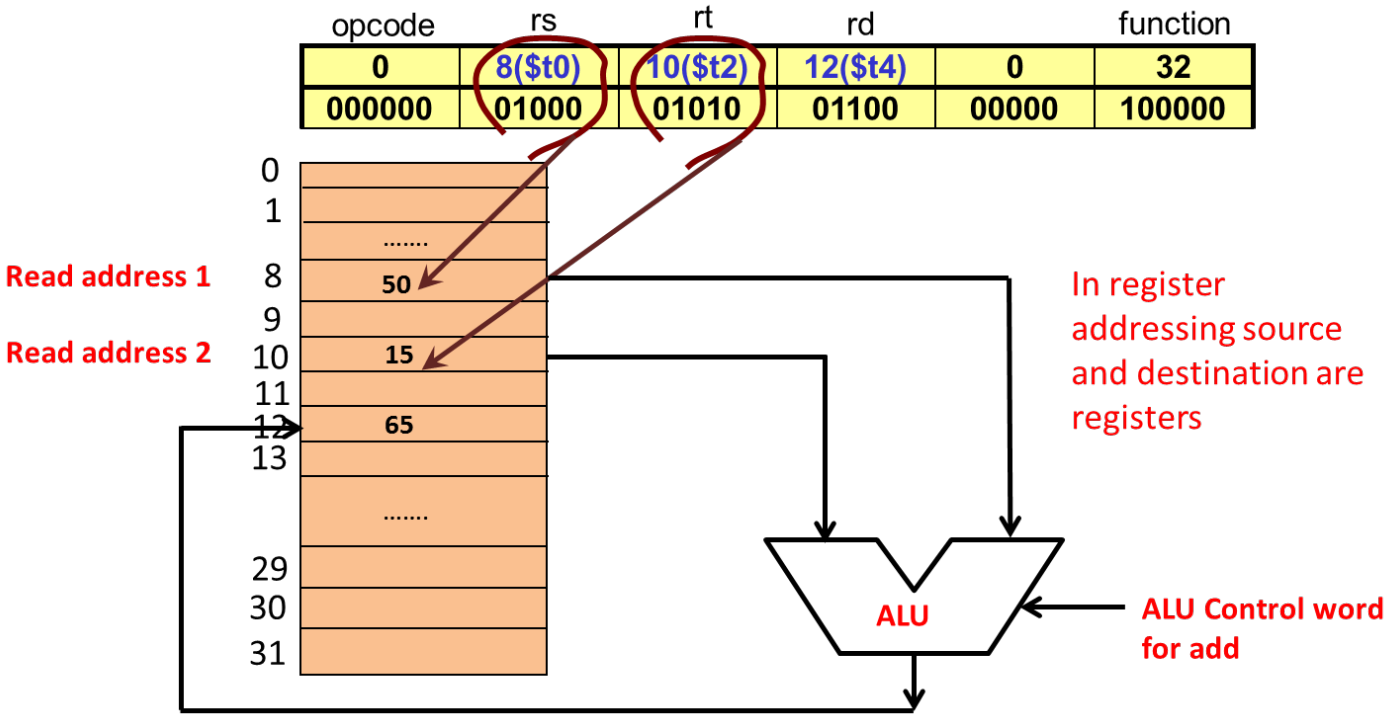


- All data values (operands/result) are located in registers.
- rs and rt : specify the first and the second source registers
- rd: specifies the destination register
- sh stands for shift amount: specifies number of bit positions to be shifted (used in shift instructions.)
- funct: function code field is an opcode extension: specifies the sub-type of instruction.

Operation	Opcode	function
ADD	000000	100000
SUB	000000	100010
AND	000000	100100
OR	000000	100101

MIPS ISA – Instruction Format

- Register (R) Type – Register Addressing
  - Instruction structure: <operation> <rd>, <rs>, <rt>
  - E.g. add $12, $8, $10, (add $t4, $t0, $t2) meaning: [R12]← [R8]+[R10]
  - It is an R-type instruction of the form: add $rd, $rs, $rt



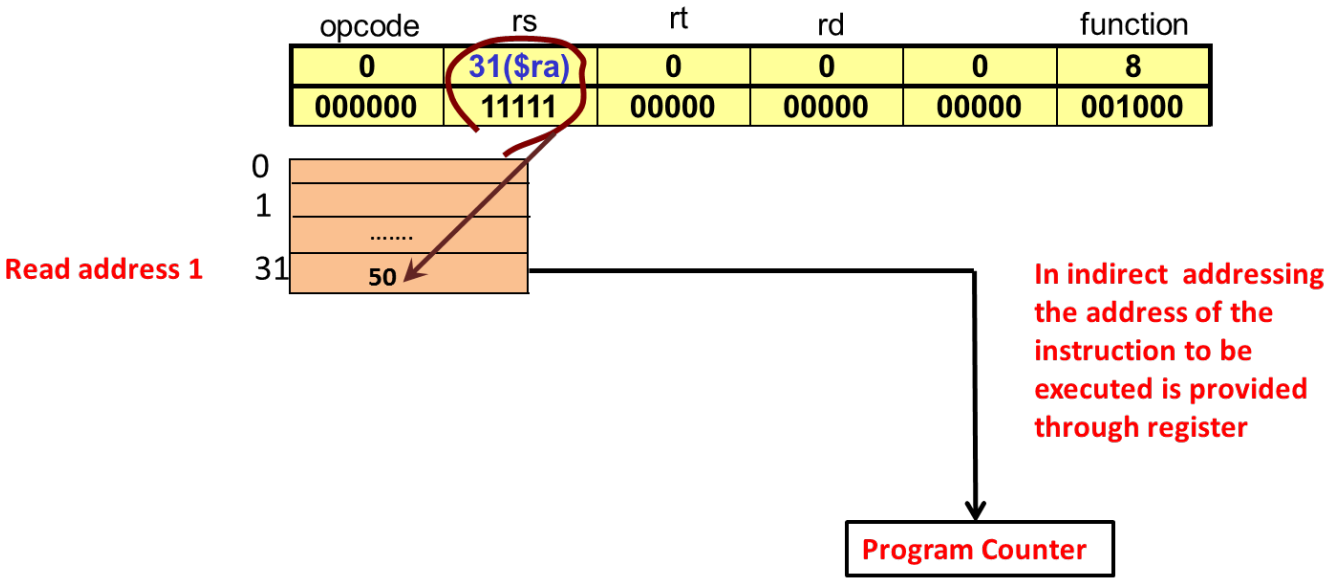
MIPS ISA – Instruction Format

- Register (R) Type – Indirect Addressing
  - JR: stands for "jump register" or jump to an address stored in a register

    Eg: JR $31, (JR $ra) meaning: Jump to the content in register $31.

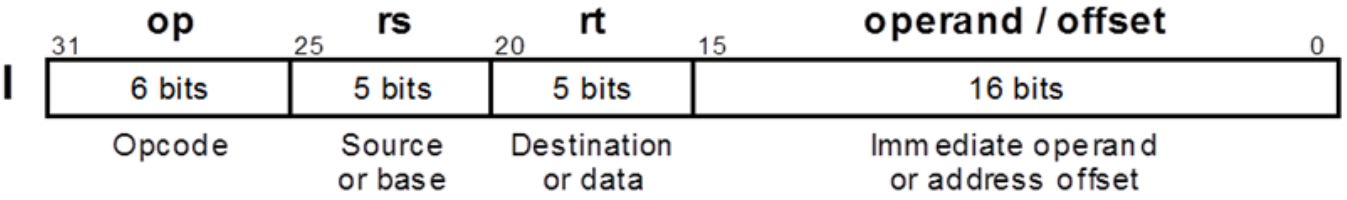
    [PC] ← [$31]

    It is an R-type instruction



MIPS ISA – Instruction Format

- Immediate (I) Type



- Class of operation that operates on an immediate value and a register value
- rs: specify the first source registers
- "immediate field": contains a 16-bit 2's complement number.
- rt: specifies the destination register

❖ Instruction structure: <operation> <rt>, <rs>, <immediate>

Example: ADD $rt, $rs, immediate

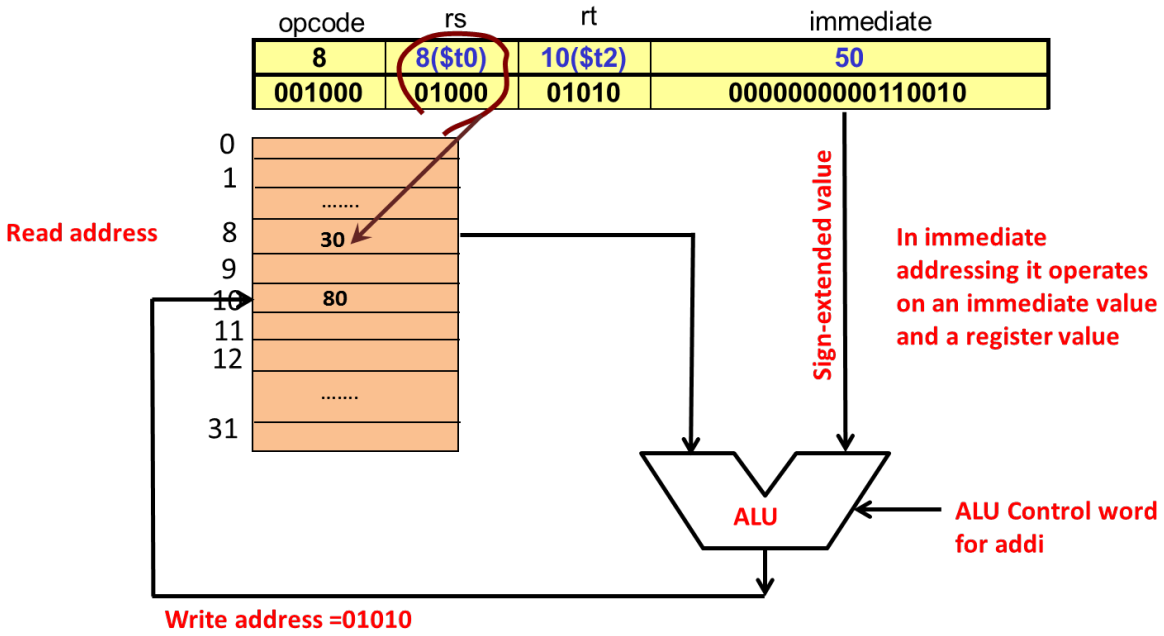
Meaning: $rt ← $rs + sign-extended(immediate): add content of register $rs with sign extended form of content of "immediate" field, and store result in destination register $rd.

❖ Note: all operands for arithmetic operations need to be in 2's complement (2C) representation.

MIPS ISA – Instruction Format

- Immediate (I) Type - Immediate Addressing
  - Example: ADD $10, $8, 50 (ADD $t2, $t0, 50): meaning [R10] ← [R8] + 50
  - It is an I-type instruction of form:

    add $rd, $rs, imm, and 16-bit operand "imm=50" in immediate field need to be sign-extended to 32-bits.

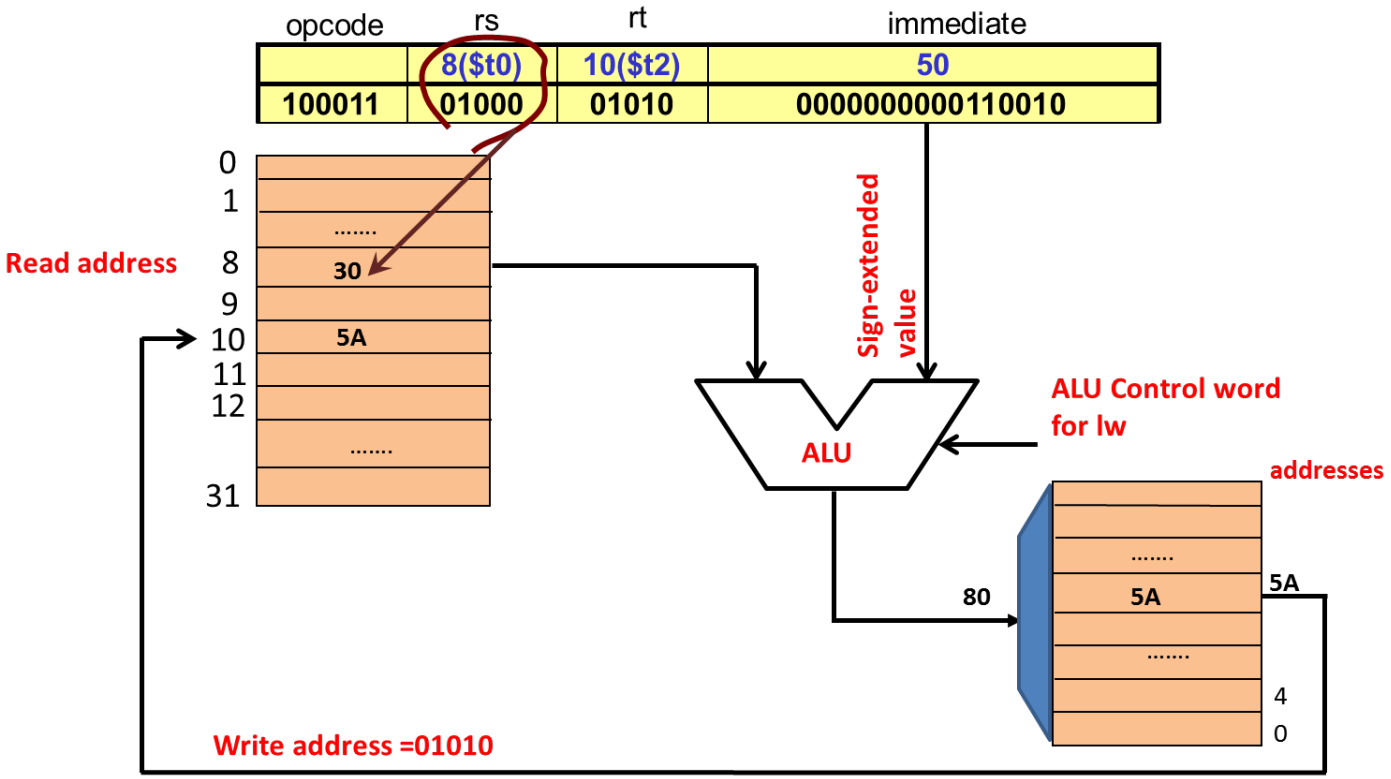


MIPS ISA – Instruction Format

- Immediate (I) Type – Memory Related Instructions
  - Memory related instructions: Load word and Store word.
  - Base address of memory location to be accessed is stored in a register ($rs).
  - To access wider range of memory we can add the base address with an immediate offset, Hence I-type instruction.
  - Syntax: LW $rt, offset($rs):
    - ALU calculates the address (offset+[$rs]).
    - Data at memory location (offset+[$rs]) is read.
    - The read data is saved in the destination register ($rt).
  - Address of the operand= "immediate" + content of source register $rs. (offset+[$rs]).
  - Example: LW $10, 50($8), meaning: [$10] ← mem[[$8]+50], i.e., load the content at memory location ([$8]+50) at $10.
  - Sign-extension is needed to extend the 16-bit offset to 32-bit offset.

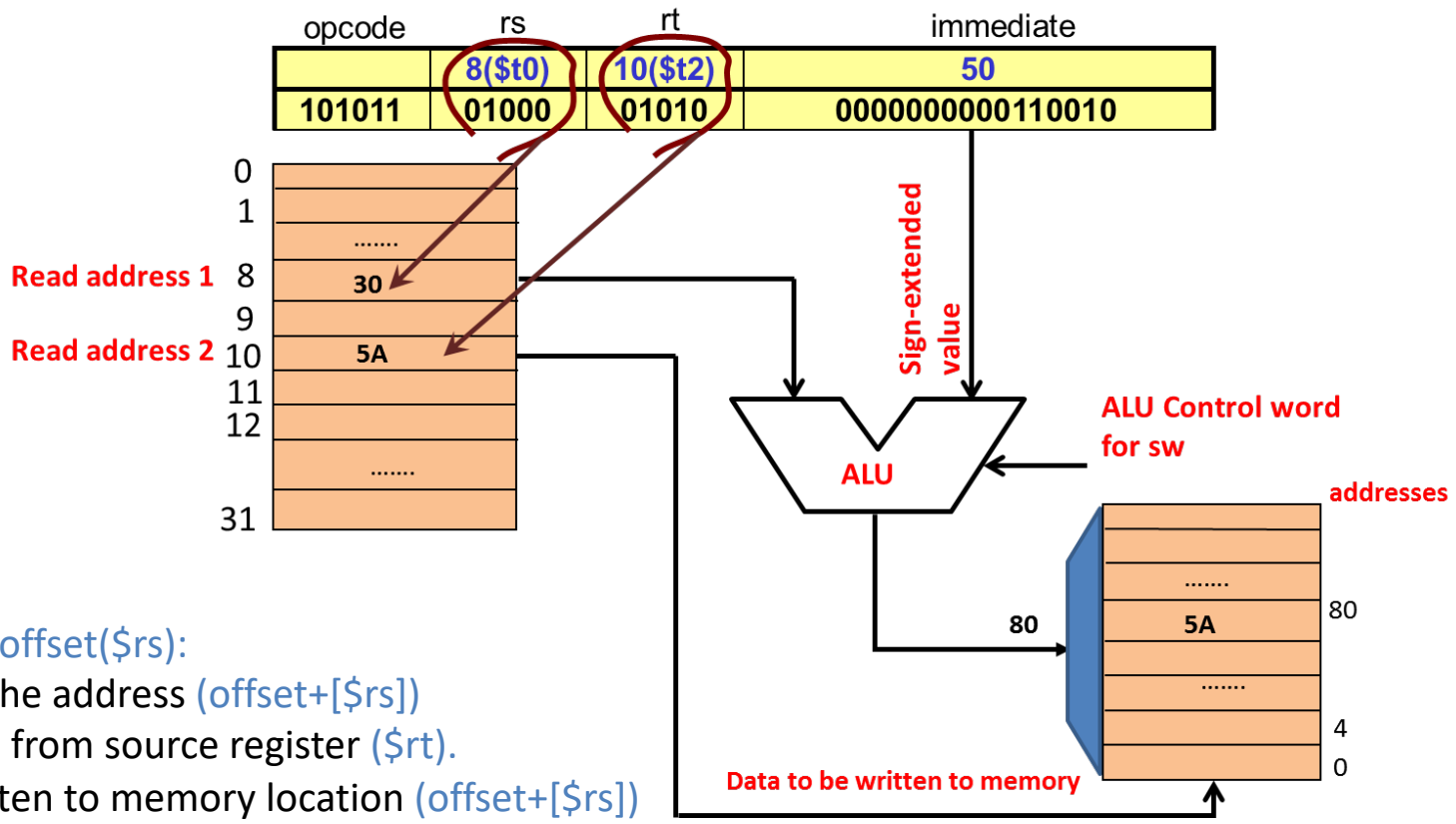
MIPS ISA – Instruction Format

- Immediate (I) Type – Base Addressing
  - LW $10, 50($8) (LW $t2, 50($t0)): meaning [R10] ← [[R8] + 50]



MIPS ISA – Instruction Format

- Immediate (I) Type – Base Addressing
  - SW $10, 50($8) (SW $t2, 50($t0)): meaning [$10] → [[$8] + 50]

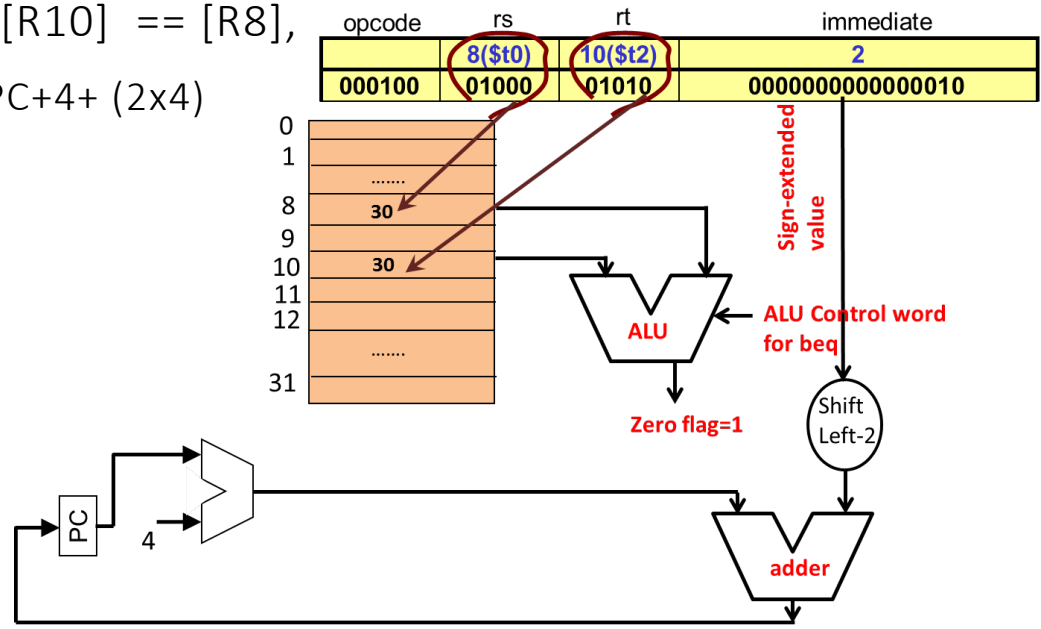


Syntax: SW $rt, offset($rs):
ALU calculates the address (offset+[$rs])
The data is read from source register ($rt).
This data is written to memory location (offset+[$rs])

MIPS ISA – Instruction Format

- Immediate (I) Type – PC Relative Addressing
- Example: BEQ $10, $8, 2: meaning [R10] == [R8], then PC will move to PC+4+ (2x4)

PC Address	Instruction mem
0x00000000	lw $3, $2(3)
0x00000004	add $3, $2,$4
0x00000008	beq $10,$8,2
0x0000000C	add $10, $5,$4
0x000000010	xor $3, $2,$4
0x000000014	sw $3, $2(3)
……	……
0xFFFFFFF4	……
0xFFFFFFF8	……
0xFFFFFFFC	……



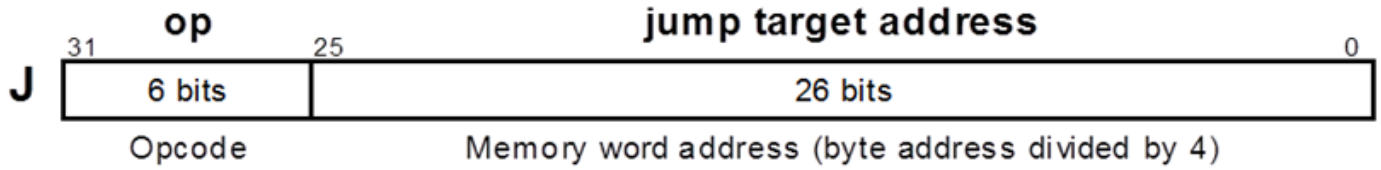
Step 1 → [R10] == [R8], then nPC=PC+4+(2x4)

Skip 2 instructions

newPC (nPC) points to this location

MIPS ISA – Instruction Format

- Jump (J) Type



- J-type instructions: used when a jump in instruction sequence is to be performed.
- For jump, we need to change content of program counter (PC),which stores address of next instruction to be executed.
- Syntax: J offset.

  meaning: [nPC] ←$[PC]_{31\text{-}28}$:: target address * 4

  $[PC]_{31\text{-}28}$ refers to the 4 upper bits of PC.
- The target address is word address and it is multiplied by 4 to make it byte addressable.
- Addressing mode: pseudo direct addressing.

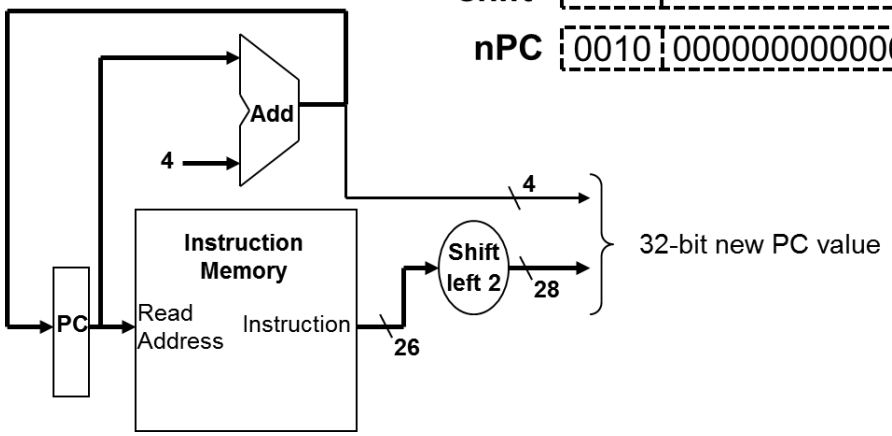
MIPS ISA – Instruction Format

- Jump (J) Type - Pseudo direct addressing

J15 – Jump to the location 15

opcode	Word Address=15
000010	00000000000000000000001111

current PC	0010	00000000000000000000000000
jump address		00000000000000000000001111
shift		00
nPC	0010	00000000000000000000001111 00



The address of next instruction is constructed by concatenating {4 upper bits of PC},{26 offset bits in the Instruction}, and {00}

26 bits in the immediate field and two bits of "shift" can be used to access a region of 228= 256 MB of the memory defined by the 4 upper bits of PC.

MIPS ISA

- Classification of Instructions based on functionality
  - **ALU operations**
    - arithmetic operations: add, sub, mult, div and their variants
    - logical operations: AND, OR, NOR, etc.
    - shift operations: sll, srl, slt, etc.
  - **Data transfer**: lw, sw, lb (load byte), sb (store byte) etc.
    - load and store: memory to register (load) and register to memory (store)
  - **Control-flow**: changes the sequence of execution of instructions
    - jump (unconditional) and branch (conditional)
  - **Others**
    - system operations: operating system call and memory management
    - floating point operations: add, mult, divide, and compare
    - decimal: add, multiply, decimal-to-character
    - string: move, compare, and stretch
    - graphics: pixel and vertex operations

MIPS ISA

- ALU – Arithmetic operations: add, sub, addi, subi and their variants

Instruction	Syntax	Meaning
add	add $rd, $rs, $rt	$rd ← $rs + $rt
subtract	sub $rd, $rs, $rt	$rd ← $rs - $rt
add unsigned	add $rd, $rs, $rt	$rd ← $rs + $rt
add immediate	addi $rt, $rs, imm	$rt ← $rs + imm
add immediate unsigned	addiu $rt, $rs, imm	$rt ← $rs + imm

Q: What is the difference between add and addu?

A: When overflow occurs in case of add and sub, result is not stored at destination but an exception is raised. addu does the same operations as add but ignore overflows and the result is stored at the destination.

MIPS ISA

- ALU – Logical operations: AND, OR, XOR, ANDI and their variants

Instruction	Syntax	Meaning
AND	and $rd, $rs, $rt	$rd ← $rs & $rt
OR	or $rd, $rs, $rt	$rd ← $rs | $rt
XOR	xor $rd, $rs, $rt	$rd ← $rs ^ $rt
AND immediate	andi $rt, $rs, imm	$rt ← $rs & imm
OR immediate	ori $rt, $rs, imm	$rt ← $rs | imm
XOR immediate	xori $rt, $rs, imm	$rt ← $rs ^ imm

MIPS ISA

- ALU – Shift operations: srl, all, sra and their variants

opcode	rs	rt	rd	Shamt (h)	function

Instruction	Syntax	Meaning
Shift left logical	sll $rd, $rs, h	$rd ← $rs <<h
Shift right logical	srl $rd, $rs, h	$rd ← $rs >> h
Shift right arithmetic	sra $rd, $rs, h	$rd ← $rs >> h

- Let $rs = $a_7a_6a_5a_4a_3a_2a_1a_0$
- The shift amount, h, is 3 bits

Operation	Y
3-bit shift right logical	0 0 0 a_7 a_6 a_5 a_4 a_3
3-bit shift right arithmetic	a_7 a_7 a_7 a_7 a_6 a_5 a_4 a_3
3-bit shift left logical	a_4 a_3 a_2 a_1 a_0 0 0 0

MIPS ISA

- ALU – Shift operations: srl, all, sra and their variants

- Why SLA is not used?
  - Sign-preserving shifting from left is different than plain shifting of bits and requires consideration of overflow error.
  - If the sign bit is preserved, i.e., no overflow, SLA is equivalent to SLL.

Instruction	Syntax	Meaning
Set on less than	slt $rd, $rs, $rt	if $rs < $rt then $rd ←1
Set on less than unsigned	sltu $rd,$rs, $rt	if $rs < $rt ,$rd ← 1 else $rd ←0
Set on less than immediate	slti $rt, $rs, imm	if $rs < imm, $rt ←1; else $rt← 0

MIPS ISA

- Data transfer operations – LW, SW, LB, SB

Instruction	Syntax	Meaning
Load word	lw $rt, imm($rs)	$rt ← mem([$rs] + imm)
Store word	sw $rt, imm($rs)	mem([$rs] + imm)← $rt
Load byte	lb $rt, imm($rs)	$rt ← mem([$rs] + imm)
Store byte	sb $rt, imm($rs)	mem([$rs] + imm)← $rt

- Control flow operations – beq, bne, bgez, bgtz, j, jr

Instruction	Syntax	Meaning
Branch when equal	beq $rs , $rt, imm	Branch if ($rs==$rt)
Branch when not equal	bne $rs , $rt, imm	Branch if ($rs!=$rt)
Branch greater than or equal to zero	bgez $rs , imm	Branch if ($rs >=0)
Branch greater than zero	bgtz $rs , imm	Branch if ($rs >0)
Jump	j offset	Jump by offset words
Jump register	jr $ra	Jump to the content of $ra
Jump and link	jal offset	Jump by offset words and saves PC+4 to $ra

ISA Design Goals

- Design goals
  - Simple microarchitecture implementation
    - → Low hardware complexity
  - High performance: execution time = IC x CPI x clock period
    - → Balanced optimization
  - Less data movement, scope for parallel and pipeline implementation
    - → Low Power
  - Easy to express high-level programs by machine language instructions
    - → Programmability
  - Programmability across generation
    - → Compatibility

ISA Design Issues

- Basic Design issues
  - Selection of operations to be executed by instructions
    - → Size of instruction set, size of opcode
  - Operand location
    - → Registers, or memory, or in both
  - Addressing mode and instruction formats
  - Deal with special registers like Program Counter (PC)
  - Register size, word size, memory address space and address size

ISA Design Issues

- Basic Design issues
  - **Registers**: number of registers, special registers…
  - **Bus Sizes**: size of data bus…
  - **Operations**: number of operands..
  - How Many Operands?
  - Whether there will be Literal Values?
  - What is the Instruction Format?
  - …

ISA Design Philosophies: CISC & RISC

- CISC: Complex Instruction Set Computing
  - CISC stands for Complex Instruction Set Computer (developed in early 60s: examples PDP-11, DEC system, Intel 80x86, and Motorola 68K series).
- RISC: Reduced Instruction Set Computing
  - RISC stands for Reduced Instruction Set Computer (relatively new: started in late 1970: 1984 MIPS developed at Stanford: early 1990s IBM POWER (Performance Optimization With Enhanced RISC) architecture.

ISA Design Issues

- CISC and RISC design policy
  - CISC was developed in early 1960s: examples PDP-11, DEC system, Intel 80x86, and Motorola 68K series).
    - Aims at a small program memory and less compiler workload.
    - Involves large instruction set comprised of complex and specialized instructions, e.g., transcendental functions (exponentiation, logarithm) and string manipulation, to have fewer instructions per task.
  - RISC was started in late 1970: 1984 MIPS developed at Stanford: early 1990s IBM POWER (Performance Optimization With Enhanced RISC) architecture.
    - All computing tasks can be performed at a small instruction set comprised of simple instructions.
    - Based on 80/20 rule: 80% of instructions use only 20% of ISA
    - Each instruction can be executed in a single clock cycle of short duration.

ISA Design Issues

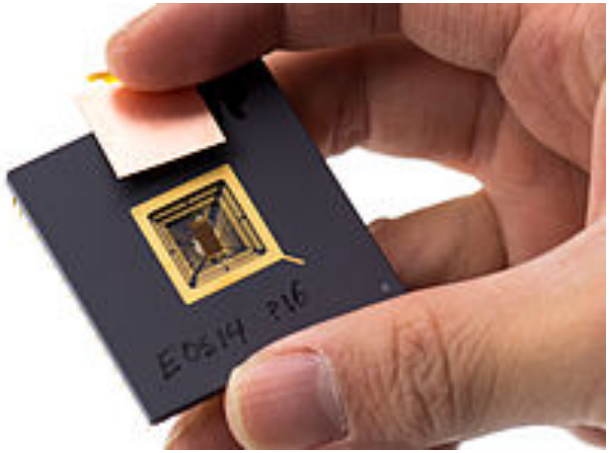
- CISC design features
  - Instructions are microcoded, executed in multiple clock cycles.
  - Program code size is relatively small.
  - Several addressing modes and multiple displacement (offset) sizes.
    - instruction length varies according to the addressing mode.
    - complex instruction-decoding logic.
  - Source and destination could be in registers/memory/ both.
    - relatively longer clock period involves less registers.
  - CISC implementations translate the instructions to RISC like microinstructions to realize pipelining.
  - Less compilation effort: complexity lies in micro-program level.

ISA Design Issues

- RISC design features - **Small and simple instruction set**

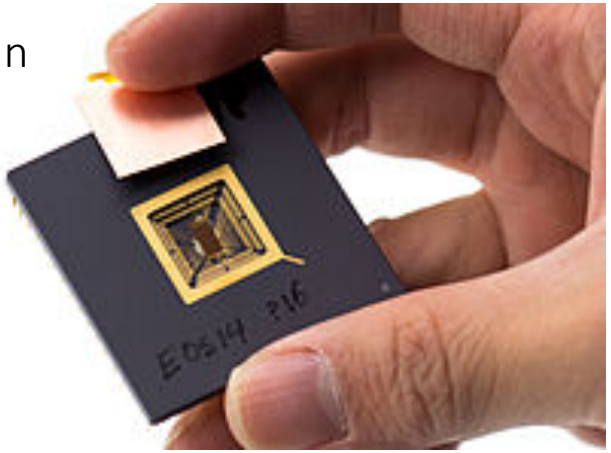
  - Fixed opcode-width and fewer addressing modes, fixed instruction-length, fewer instruction formats with common fields.

    - Advantages: simple and fast hardwired decoding and control generation, shorter clock period, load/store architecture.



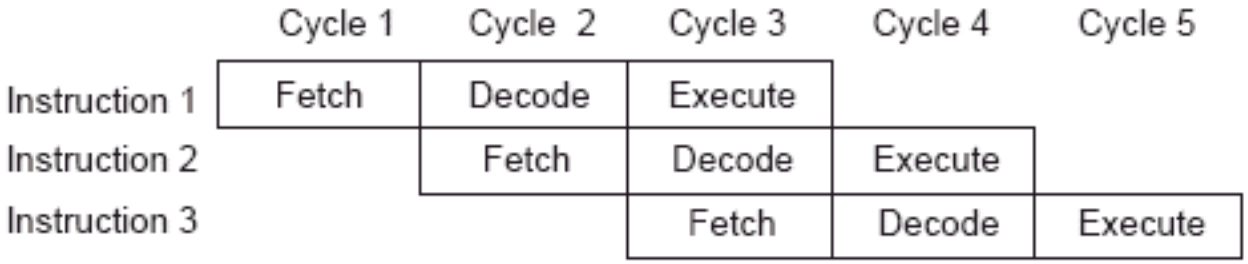
ISA Design Issues

- RISC design features - **Small and simple instruction set**
  - Only two instructions (load & store) and only one addressing mode for memory access: all operands and destination are located in registers.
    - Advantages: memory access and operand processing are performed by separate instructions: register access is faster) shorter clock period single-cycle implementation.
  - All instructions will be executed in a single cycle.
    - Advantages: makes the superscalar/instruction level pipelining simpler.



ISA Design Issues

- Microcoding



$R_i = R_j + R_k$

opcode[1]: $ri_{sel}, rj_{sel}, rk_{sel}, alu_{sel} = ADD;$

opcode[2]: $ri_{sel}, rj_{sel}, rk_{sel}, alu_{sel} = SUB;$

opcode[3]: $ri_{sel}, rj_{sel}, rk_{sel}, alu_{sel} = SHIFT;$

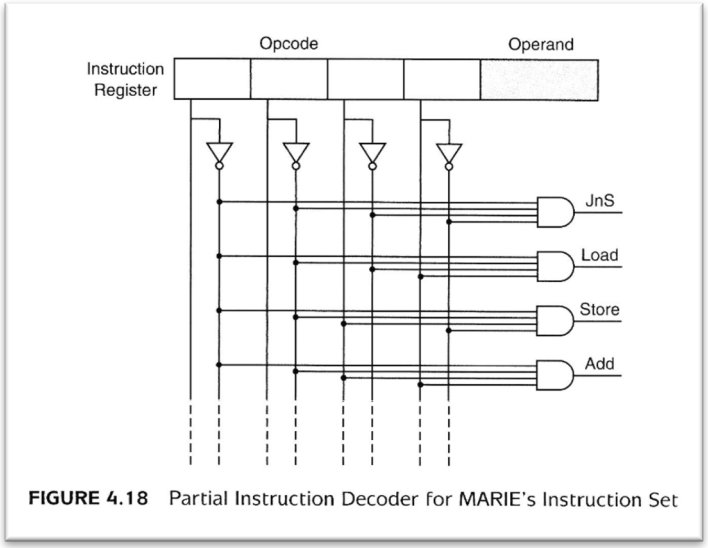


FIGURE 4.18 Partial Instruction Decoder for MARIE's Instruction Set

ISA Design Issues - CISC V.S. RISC

CISC	RISC
Many complex instructions in instruction set	Few simple instructions in instruction set
Many formats & several addressing modes to support complex instructions	Few instruction formats & addressing modes
Instruction length varies according to the addressing mode	Fixed instruction length → simple implementation
Instructions are microcoded and executed in multiple clock cycles	Hardwired decoding: single cycle instruction execution
Memory can be referenced by many different modes	Only load/store instructions can reference memory
Operands could be memory; higher clock period; less number of registers	Operands in register for faster clocking; more registers; less memory access
Difficult to pipeline and super-scalar implementation	Easy to pipeline: and super-scalar implementation
Program code size is relatively small: complexity is in micro-program level	Program code size is usually large: complexity is in the compiler
Higher complexity of instruction implementation: CPI more than 1.	More compile time: higher register and more cache area

RISC-V

- 5th major ISA to come out of UC. Berkeley in 2010
- Open standard, allowing:
  - companies to design processors royalty free
  - open source core development
  - ISA exploration and innovation
- Has industry support with folks from Bluespec, Google, Microsemi (Microchip), NVIDIA, NXP, and Western Digital
- Cleaned up ISA taking lessons learned from 30+ years of microarchitecture development

ISA - ROAD TO FUTURE

- Next generation Architecture Technology
  - EPIC is the next generation technology (e.g., RISC, CISC)

EPIC

- Explicitly
- Parallel
- Instruction
- Computing

SuperScolar

- H/W detects implicit parallelism
- H/W scheduling & speculation
- H/W renames 8-32 registers to 64+

RISC

- Simple, fixed length instructions
- Sequencing done by compiler

CISC

- Complex, variable length instructions
- Sequencing done in hardware

Performance

Time

2019

Goodbye x86.
The FUTURE is RISC-V

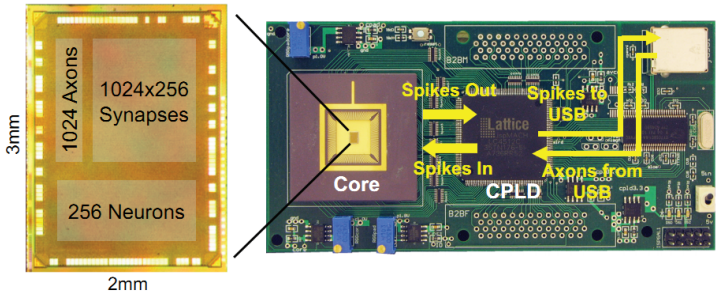
ISA - Road to Future

- RISC-V and Linux on RISC-V

- 55th DAC | Design Automation Conference 2018

What did we Learn So far

- Instruction Set Architecture
  - ✓ MIPS ISA examples
  - ✓ Details of register type instructions
- MIPS ISA
  - ✓ Immediate-type instructions
  - ✓ Control-flow instructions
  - ✓ Arithmetic instructions
  - ✓ Pseudo instructions
- Design issues
  - ✓ RISC v.s. CISC



Instruction Set Architecture (ISA)